Appendix B:

Persistence

Overview

Tools.h++ version 7.0 Users Guide, 1996, Rogue Wave Software, defines that a object can have one of four levels of persistence:

- No persistence. There is no mechanism for storage and retrieval of the object.
- Simple persistence. A level of persistence that provides storage and retrieval of individual objects to and from a stream or file. Simple persistence does not preserve pointer relationships among the persisted objects.
- Isomorphic persistence. A level of persistence that preserves the pointer relationships among the persisted objects.
- Polymorphic persistence. The highest level of persistence. Polymorphic persistence preserves pointer relationships among the persisted objects and allows the restoring process to restore an object without prior knowledge of that object's type.

This appendix provides information about the use of Isomorphic persistence through descriptions, examples, and procedures for designing persistent classes. To implement other levels of persistence it is recommended that the reader consult the relevant Tools.h++ manual pages.

Persistence Mechanism

Isomorphic persistence is the storage and retrieval of objects to and from a stream such that the pointer relationships between the objects are preserved. If there are no pointer relationships, isomorphic persistence effectively saves and restores objects the same way as simple persistence. When a collection is isomorphically persisted, all objects within that collection are assumed to have the same type.

The isomorphic persistence mechanism uses a table to keep track of pointers it has saved. When the isomorphic persistence mechanism encounters a pointer to an unsaved object, it copies the object data, saves that object data NOT the pointer to the stream, then keeps track of the pointer in the save table. If the isomorphic persistence mechanism later

encounters a pointer to the same object, instead of copying and saving the object data, the mechanism saves the save table's reference to the pointer.

When the isomorphic persistence mechanism restores pointers to objects from the stream, the mechanism uses a restore table to reverse the process. When the isomorphic persistence mechanism encounters a pointer to an unrestored object, it recreates the object with data from the stream, then changes the restored pointer to point to the recreated object. The mechanism keeps track of the pointer in the restore table. If the isomorphic persistence mechanism later encounters a reference to an already-restored pointer, then the mechanism looks up the reference in the restore table, and updates the restored pointer to point to the object referred to in the table.

Class Requirements For Persistence

To create a class that supports isomorphic persistence the class must meet the following requirements.

- The class must have appropriate default and copy constructors defined or generated by the compiler:

```
PClass();      // default constructor
PClass(T& t); // copy constructor
```

- The class must have an assignment operator defined as a member OR as a global function:

```
PClass& operator=(const PClass& pc); // member function
PClass& operator=(PClass& lhs, const PClass& rhs); // global function
```

- The class cannot have any non-type template parameters. For example, in RWTBitVec<size>, "size" is placeholder for a value rather than a type. No present compiler accepts function templates with non-type template parameters, and the global functions used to implement isomorphic persistence (rwRestoreGuts and RWSaveGuts) are function templates when they are used to persist templatized classes.

- All the data necessary to recreate an instance of the class must be globally available (have accessor functions).

Creating a Persistent Class

To create an isomorphically persistent class or to add isomorphic persistence to an existing class, follow these steps:

1. Make all necessary class data available.

2. Add RWDECLARE_PERSISTABLE to your header file.

```
#include <rw/edefs.h>
RWDECLARE_PERSISTABLE(YourClass)
```

3. Add RWDEFINE_PERSISTABLE to one source file.

```
#include <rw/epersist.h>
RWDEFINE_PERSISTABLE(YourClass)
```

4. Define rwSaveGuts and rwRestoreGuts. Methods rwSaveGuts and rwRestoreGuts will be used to save and restore the internal state of the class. These methods are called by the operator<< and operator>> that were declared and defined by the macros in 2 & 3.

For non-templatized classes , define the following functions:

```
void rwSaveGuts(RWFile& f, const YourClass& t){/*_*/}
void rwSaveGuts(RWvostream& s, const YourClass& t) {/*_*/}
void rwRestoreGuts(RWFile& f, YourClass& t) {/*_*/}
void rwRestoreGuts(RWvistream& s, YourClass& t) {/*_*/}
```

For templatized classes with a single template parameter T, define the following functions:

```
template<class T> void
rwSaveGuts(RWFile& f, const YourClass<T>& t){/*_*/}
template<class T> void
rwSaveGuts(RWvostream& s, const YourClass<T>& t) {/*_*/}
template<class T> void
rwRestoreGuts(RWFile& f, YourClass<T>& t) {/*_*/}
template<class T>void
```

rwRestoreGuts(RWvistream& s, YourClass<T>& t) {/*_*/}

For templatized classes with more than one template parameter, define rwRestoreGuts and rwSaveGuts with the appropriate number of template parameters.

Function rwSaveGuts saves the state of each class member necessary persistence to an RWvostream or an RWFile. If the members of your class can be persisted and if the necessary class members are accessible to rwSaveGuts, you can use operator<< to save the class members.

Function rwRestoreGuts restores the state of each class member necessary for persistence from an RWvistream or an RWFile. Provided that the members of your class are types that can be persisted, and provided that the members of your class are accessible to rwRestoreGuts, you can use operator>> to restore the class members.

Example of a Persistent Class
PClass Header File

```
#include <rw/cstring.h>
#include <rw/edefs.h>
#include <rw/rwfile.h>
#include <rw/epersist.h>

class PClass
{
        public:

                PClass ();
                PClass (const RWCString& string_attribute,
                                int int_attribute,
                                float float_attribute,
                                PClass* ptr_to_attribute);

                ~PClass();

                // Persistance operations
                friend void rwRestoreGuts(RWvistream& is, PClass& obj);
```

```
PClass::~PClass()
{
}

RWDEFINE_PERSISTABLE(PClass)

void rwRestoreGuts(RWvistream& is, PClass& obj)
{
        is >> obj.StringAttribute;    // Restore String.
        is >> obj.IntAttribute;       // Restore Int.
        is >> obj.FloatAttribute;     // Restore Float.

        RWBoolean ptr;
        is >> ptr;
        if (ptr)
        {
                        is >> obj.PtrToAttribute;
        }
}

void rwRestoreGuts(RWFile& file, PClass& obj)
{
        file >> obj.StringAttribute;  // Restore String.
        file >> obj.IntAttribute;     // Restore Int.
        file >> obj.FloatAttribute;   // Restore Float.

        RWBoolean ptr;
        file >> ptr;
        if (ptr)
        {
                file >> obj.PtrToAttribute;
        }
}

void rwSaveGuts(RWvostream& os, const PClass& obj)
{
        os << obj.StringAttribute;   // Save String.
```

```
        friend void rwRestoreGuts(RWFile& file, PClass& obj);
        friend void rwSaveGuts(RWvostream& os, const PClass& obj);
        friend void rwSaveGuts(RWFile& file, const PClass& obj);

        // Stream operations
        friend ostream& operator<<(ostream& os, const PClass& obj);

    private:

        RWCString StringAttribute;
        int      IntAttribute;
        float    FloatAttribute;
        PClass*  PtrToAttribute;
};

RWDECLARE_PERSISTABLE(PClass)
```

PClass Implementation File

```
#include <PClass.H>

PClass::PClass()
{
    IntAttribute = 0;
    FloatAttribute = 0;
    PtrToAttribute = 0;
}

PClass::PClass(const RWCString& string_attribute,
               int int_attribute,
               float float_attribute,
               PClass* ptr_to_attribute)
{
    StringAttribute = string_attribute;
    IntAttribute = int_attribute;
    FloatAttribute = float_attribute;
    PtrToAttribute = ptr_to_attribute;
}
```

```
        os << obj.IntAttribute;        // Save Int.
        os << obj.FloatAttribute;      // Save Float.

        if (obj.PtrToAttribute == rwnil)
        {
        os << FALSE;                   // No pointer.
        }
        else
        {
        os << TRUE;                    // Save Pointer
        os << *(obj.PtrToAttribute);
        }
}

void rwSaveGuts(RWFile& file, const PClass& obj)
{
        file << obj.StringAttribute;  // Save String.
        file << obj.IntAttribute;      // Save Int.
        file << obj.FloatAttribute;    // Save Float.

        if (obj.PtrToAttribute == rwnil)
        {
        file << FALSE;                 // No pointer.
        }
        else
        {
        file << TRUE;                  // Save Pointer
        file << *(obj.PtrToAttribute);
        }
}

ostream& operator<<(ostream& os, const PClass& obj)
{
        os << "\nStringAttribute : "
           << obj.StringAttribute << "\n";

        os << "IntAttribute : "
           << obj.IntAttribute << "\n";
```

```
        os << "FloatAttribute : "
           << obj.FloatAttribute << "\n";

        os << "PtrToAttribute : "
           << (void*)obj.PtrToAttribute << "\n";

        if (obj.PtrToAttribute)
        {
        os << "Value at Pointer : "
           << *(obj.PtrToAttribute) << "\n";
        }

        return os;
}

Use of PClass
#include <iostream.h>
#include <PClass.H>

void main()
{
        // Create object that will be pointed to by
        // persistent object.
        RWCString s1("persist_pointer_object");
        PClass persist_pointer_object(s1, 1, 1.0, 0);

        RWCString s2("persist_class1");
        PClass persist_class1(s2, 2, 2.0, &persist_pointer_object);

        cout << "persist_class1 (before save):" << endl
           << persist_class1 << endl << endl;

        // Save object in file "test.dat".
        RWFile file("test.dat");
        file << persist_class1;

        PClass persist_class2;
```

```
// Restore object from file "test.dat".
{
RWFile file("test.dat");
file >> persist_class2;
}

cout << "persist_class2 (after restore):" << endl
    << persist_class2 << endl << endl;
}
```

Special Care with Persistence

The persistence mechanism is a useful quality, but requires care in some areas. Here are a few things to look out for when using persist classes.

1. Always Save an Object by Value before Saving the Identical Object by Pointer.

In the case of both isomorphic and polymorphic persistence of objects, never stream out an object by pointer before streaming out the identical object by value. Whenever designing a class that contains a value and a pointer to that value, the saveGuts and restoreGuts member functions for that class should always save or restore the value then the pointer.

2. Don't Save Distinct Objects with the Same Address.

Be careful not to isomorphically save distinct objects that may have the same address. The internal tables that are used in isomorphic and polymorphic persistence use the address of an object to determine whether or not an object has already been saved.

3. Don't Use Sorted RWCollections to Store Heterogeneous RWCollectables.

When you have more than one different type of RWCollectable stored in an RWCollection, you can't use a sorted RWCollection. For example, this means that if you plan to store RWCollectableStrings and RWCollectableDates in the same RWCollection, you can't store them in a sorted RWCollection such as RWBtree. The sorted RWCollections are

RWBinaryTree, RWBtree, RWBTreeDictionary, and RWSortedVector. The reason for this restriction is that the comparison functions for sorted RWCollections expect that the objects to be compared will have the same type.

4. Define All RWCollectables That Will Be Restored.

These declarations are of particular concern when you save an RWCollectable in a collection, then attempt to take advantage of polymorphic persistence by restoring the collection in a different program, without using the RWCollectable that you saved. If you don't declare the appropriate variables, during the restore attempt the RWFactory will throw an RW_NOCREATE exception for some RWCollectable class ID that you know exists. The RWFactory won't throw an RW_NOCREATE exception when you declare variables of all the RWCollectables that could be polymorphically restored.

The problem occurs because the compiler's linker only links the code that RWFactory needs to create the missing RWCollectable when that RWCollectable is specifically mentioned in your code. Declaring the missing RWCollectables gives the linker the information it needs to link the appropriate code needed by RWFactory.